

Control Improvisation

Daniel J. Fremont, Alexandre Donzé, Sanjit A. Seshia, and David Wessel

University of California, Berkeley

Abstract

We formalize and analyze a new automata-theoretic problem termed *control improvisation*. Given an automaton, the problem is to produce an *improviser*, a probabilistic algorithm that randomly generates words in its language, subject to two additional constraints: the satisfaction of an *admissibility* predicate, and the exhibition of a specified amount of randomness. Control improvisation has multiple applications, including, for example, generating musical improvisations that satisfy rhythmic and melodic constraints, where admissibility is determined by some bounded divergence from a reference melody. We analyze the complexity of the control improvisation problem, giving cases where it is efficiently solvable and cases where it is $\#P$ -hard or undecidable. We also show how symbolic techniques based on Boolean satisfiability (SAT) solvers can be used to approximately solve some of the intractable cases.

1998 ACM Subject Classification F.4.3 Formal Languages, G.3 Probability and Statistics, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases finite automata, random sampling, Boolean satisfiability, testing, computational music, control theory

1 Introduction

We introduce and formally characterize a new automata-theoretic problem termed *control improvisation*. Given an automaton, the problem is to produce an *improviser*, a probabilistic algorithm that randomly generates words in the language of the automaton, subject to two additional constraints: each generated word must satisfy an *admissibility* predicate, and the improviser must exhibit a specified amount of randomness.

The original motivation for this problem arose from a topic known as *machine improvisation of music* [19]. Here, the goal is to create algorithms which can generate variations of a reference melody like those commonly improvised by human performers, for example in jazz. Such an algorithm should have three key properties. First, the melodies it generates should conform to rhythmic and melodic constraints typifying the music style (e.g. in jazz, the melodies should follow the harmonic conventions of that genre). Second, the algorithm should be sufficiently randomized that running it several times produces a variety of different improvisations. Finally, the generated melodies should be actual variations on the reference melody, neither reproducing it exactly nor being so different as to be unrecognizable. In previous work [11], we identified these properties in an initial definition of the control improvisation problem, and applied it to the generation of monophonic (solo) melodies over a given jazz song harmonization¹.

These three properties of a generation algorithm are not specific to music. Consider *black-box fuzz testing* [20], which produces many inputs to a program hoping to trigger a bug. Often, constraints are imposed on the generated inputs, e.g. in *generative fuzz testing*

¹ Examples of improvised melodies can be found at the following URL:
http://www.eecs.berkeley.edu/~donze/impro_page.html.



approaches which enforce an appropriate format so that the input is not rejected immediately by a parser. Also common are *mutational* approaches which guide the generation process with a set of real-world seed inputs, generating only inputs which are variations of those in the set. And of course, fuzzers use randomness to ensure that a variety of inputs are tried. Thus we see that the inputs generated in fuzz testing have the same general requirements as music improvisations: satisfying a set of constraints, being appropriately similar/dissimilar to a reference, and being sufficiently diverse.

We propose control improvisation as a precisely-defined theoretical problem capturing these requirements, which are common not just to the two examples above but to many other generation problems. Potential applications also include home automation mimicking typical occupant behavior (e.g., randomized lighting control obeying time-of-day constraints and limits on energy usage [16]) and randomized variants of the supervisory control problem [5], where a controller keeps the behavior of a system within a safe operating region (the language of an automaton) while adding diversity to its behavior via randomness. A typical example of the latter is surveillance: the path of a patrolling robot should satisfy various constraints (e.g. not running into obstacles) and be similar to a predefined route, but incorporate some randomness so that its location is not too predictable [15].

Our focus, in this paper, is on the *theoretical characterization of control improvisation*. Specifically, we give a precise theoretical definition and a rigorous characterization of the complexity of the control improvisation problem under various conditions on the inputs to the problem. While the problem is distinct from any other we have encountered in the literature, our methods are closely connected to prior work on random sampling from the languages of automata and grammars [13, 10, 14], and sampling from the satisfying assignments of a Boolean formula [7]. Probabilistic programming techniques [12] could be used for sampling under constraints, but the present methods cannot be used to construct improvisers meeting our definition.

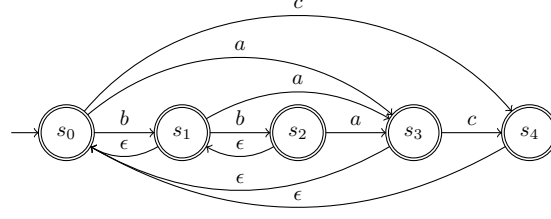
In summary, this paper makes the following novel contributions:

- Formal definitions of the notions of control improvisation (CI) and a polynomial-time improvisation scheme (Sec. 2);
- A theoretical characterization of the conditions under which improvisers exist (Sec. 3);
- A polynomial-time improvisation scheme for a practical class of CI instances, involving finite-memory admissibility predicates (Sec. 4);
- #P-hardness and undecidability results for more general classes of the problem (Sec. 5);
- A symbolic approach based on Boolean satisfiability (SAT) solving that is useful in the case when the automata are finite-state but too large to represent explicitly (Sec. 6).

We conclude in Sec. 7 with a synopsis of results and directions for future work. For lack of space, we include only selected proofs and proof sketches in the main body of the paper; complete details may be found in the Appendix.

2 Background and Problem Definition

In this section, we first provide some background on a previous automata-theoretic method for music improvisation based on a data structure called the *factor oracle*. We then provide a formal definition of the control improvisation problem while explaining the choices made in this definition.



■ **Figure 1** Factor oracle constructed from the word $w_{\text{ref}} = bbac$.

2.1 Factor Oracles

An effective and practical approach to machine improvisation of music (used for example in the prominent OMax system [1]) is based on a data structure called the factor oracle [2, 8]. Given a word w_{ref} of length N that is a symbolic encoding of a reference melody, a factor oracle F is an automaton constructed from w_{ref} with the following key properties: F has $N + 1$ states, all accepting, chained linearly with direct transitions labelled with the letters in w_{ref} , and with potentially additional forward and backward transitions. Figure 1 depicts F for $w_{\text{ref}} = bbac$. A word w accepted by F consists of concatenated “factors” of w_{ref} , and its dissimilarity with w_{ref} is correlated with the number of non-direct transitions. By assigning a small probability α to non-direct transitions, F becomes a generative Markov model with tunable “divergence” from w_{ref} . In order to impose more musical structure on the generated words, our previous work [11] additionally requires that improvisations satisfy rules encoded as deterministic finite automata, by taking the product of the generative Markov model and the DFAs. While this approach is heuristic and lacks any formal guarantees, it has the basic elements common to machine improvisation schemes: (i) it involves randomly generating strings from a formal language typically encoded as an automaton, (ii) it enforces diversity in the generated strings, and (iii) it includes a requirement on which strings are admissible based on their divergence from a reference string. The definition we propose below captures these elements in a rigorous theoretical manner, suitable for further analysis. In Sec. 4, we revisit the factor oracle, sketching how the notion of divergence from w_{ref} that it represents can be encoded in our formalism.

2.2 Problem Definition

We abbreviate deterministic and nondeterministic finite automata as DFAs and NFAs respectively. We use the standard definition of probabilistic finite automata from [18], where a string is accepted iff it causes the automaton to reach an accepting state with probability greater than a specified *cut-point* $p \in [0, 1)$. We call a probabilistic finite automaton, together with a choice of cut-point so that its language is definite, a PFA. We write $\Pr[f(X) \mid X \leftarrow D]$ for the probability of event $f(X)$ given that the random variable X is drawn from the distribution D .

► **Definition 2.1.** An *improvisation automaton* is a finite automaton (DFA, NFA, or PFA) \mathcal{I} over a finite alphabet Σ . An *improvisation* is any word $w \in L(\mathcal{I})$, and $I = L(\mathcal{I})$ is the set of all improvisations.

► **Definition 2.2.** An *admissibility predicate* is a computable predicate $\alpha : \Sigma^* \rightarrow \{0, 1\}$. An improvisation $w \in I$ is *admissible* if $\alpha(w) = 1$. We write A for the set of all admissible improvisations.

Running Example. Our concepts will be illustrated with a simple example. Our aim is to produce variations of the binary string $s = 001$ of length 3, subject to the constraint that there cannot be two consecutive 1s. So $\Sigma = \{0, 1\}$, and \mathcal{I} is a DFA which accepts all length-3 strings that do not have two 1s in a row. To ensure that our variations are similar to s , we let our admissibility predicate $\alpha(w)$ be 1 if the Hamming distance between w and s is at most 1, and 0 otherwise. Then the improvisations are the strings 000, 001, 010, 100, and 101, of which 000, 001, and 101 are admissible.

Intuitively, an improviser samples from the set of improvisations according to some distribution. But what requirements must one impose on this distribution? Since we want a variety of improvisations, we require that each one is generated with probability at most some bound ρ . By choosing a small value of ρ we can thus ensure that many different improvisations can be generated, and that no single one is output too frequently. Other constraints are possible, e.g. requiring that every improvisation have nonzero probability, but we view this as too restrictive: if there are a large number of possible improvisations, it should be acceptable for an improviser to generate many but not all of them. Another possibility would be to ensure variety by imposing some minimum distance between the improvisations. This could be reasonable in a setting (such as music) where there is a natural metric on the space of improvisations, but we choose to keep our setting general and not assume such a metric. Finally, we require our generated improvisation to be admissible with probability at least $1 - \epsilon$ for some specified ϵ . When the admissibility predicate encodes a notion of similarity to a reference string, for example, this allows us to require that our improvisations usually be similar to the reference. Combining these requirements, we obtain our definitions of an acceptable distribution over improvisations and thus of an improviser:

► **Definition 2.3.** Given $\mathcal{C} = (\mathcal{I}, \alpha, \epsilon, \rho)$ with \mathcal{I} and α as in Definitions 2.1 and 2.2, $\epsilon \in [0, 1] \cap \mathbb{Q}$ an error probability, and $\rho \in (0, 1] \cap \mathbb{Q}$ a probability bound, a distribution $D : \Sigma^* \rightarrow [0, 1]$ with support S is an (ϵ, ρ) -improvising distribution if:

- $S \subseteq I$
- $\forall w \in S, D(w) \leq \rho$
- $\Pr[w \in A \mid w \leftarrow D] \geq 1 - \epsilon$

If there is an (ϵ, ρ) -improvising distribution, we say that \mathcal{C} is (ϵ, ρ) -feasible (or simply *feasible*). An (ϵ, ρ) -improviser (or simply *improviser*) for a feasible \mathcal{C} is an expected finite-time probabilistic algorithm generating strings in Σ^* whose output distribution (on empty input) is an (ϵ, ρ) -improvising distribution.

To summarize, if $\mathcal{C} = (\mathcal{I}, \alpha, \epsilon, \rho)$ is feasible, there exists a distribution satisfying the requirements in Definition 2.3, and an improviser is a probabilistic algorithm for sampling from one.

Running Example. For our running example, $\mathcal{C} = (\mathcal{I}, \alpha, 0, 1/4)$ is not feasible since $\epsilon = 0$ means we can only generate admissible improvisations, and since there are only 3 of those we cannot possibly give them all probability at most $1/4$. Increasing ρ to $1/3$ would make \mathcal{C} feasible. Increasing ϵ to $1/4$ would also work, allowing us to return an inadmissible improvisation $1/4$ of the time: an algorithm uniformly sampling from $\{000, 001, 101, 100\}$ would be an improviser for $(\mathcal{I}, \alpha, 1/4, 1/4)$.

► **Definition 2.4.** Given $\mathcal{C} = (\mathcal{I}, \alpha, \epsilon, \rho)$, the *control improvisation (CI)* problem is to decide whether \mathcal{C} is feasible, and if so to generate an improviser for \mathcal{C} .

Ideally, we would like an efficient algorithm to solve the CI problem. Furthermore, the improvisers our algorithm produces should themselves be efficient, in the sense that their runtimes are polynomial in the size of the original CI instance. This leads to our last definition:

► **Definition 2.5.** A *polynomial-time improvisation scheme* for a class \mathcal{P} of CI instances is a polynomial-time algorithm S with the following properties:

- for any $\mathcal{C} \in \mathcal{P}$, if \mathcal{C} is feasible then $S(\mathcal{C})$ is an improviser for \mathcal{C} , and otherwise $S(\mathcal{C}) = \perp$
- there is a polynomial $p : \mathbb{R} \rightarrow \mathbb{R}$ such that if $G = S(\mathcal{C}) \neq \perp$, then G has expected runtime at most $p(|\mathcal{C}|)$.

A polynomial-time improvisation scheme for a class of CI instances is an efficient, uniform way to solve the control improvisation problem for that class. In Sections 4 and 5 we will investigate which classes have such improvisation schemes.

3 Existence of Improvisers

It turns out that the feasibility of an improvisation problem is completely determined by the sizes of I and A :

► **Theorem 3.1.** For any $\mathcal{C} = (\mathcal{I}, \alpha, \epsilon, \rho)$, the following are equivalent:

- (a) \mathcal{C} is feasible.
- (b) $|I| \geq 1/\rho$ and $|A| \geq (1 - \epsilon)/\rho$.
- (c) There is an improviser for \mathcal{C} .

Proof. (a) \Rightarrow (b): Suppose D is an (ϵ, ρ) -improvising distribution with support S . Then $\rho|S| = \sum_{w \in S} \rho \geq \sum_{w \in S} D(w) = 1$, so $|I| \geq |S| \geq 1/\rho$. We also have $\rho|S \cap A| = \sum_{w \in S \cap A} \rho \geq \sum_{w \in S \cap A} D(w) = \Pr[w \in A \mid w \leftarrow D] \geq 1 - \epsilon$, so $|A| \geq |S \cap A| \geq (1 - \epsilon)/\rho$.

(b) \Rightarrow (c): Defining $N = \lceil (1 - \epsilon)/\rho \rceil$, we have $|A| \geq N$. If $N \geq 1/\rho$, then there is a subset $S \subseteq A$ with $|S| = \lceil 1/\rho \rceil$. Since $1/\lceil 1/\rho \rceil \leq \rho$, the uniform distribution on S is a $(0, \rho)$ -improvising distribution. Since this distribution has finite support and rational probabilities, there is an expected finite-time probabilistic algorithm sampling from it, and this is a $(0, \rho)$ -improviser. If instead $N < 1/\rho$, defining $M = \lceil 1/\rho \rceil - N$ we have $M \geq 1$. Since $|I| \geq \lceil 1/\rho \rceil = N + M$, there are disjoint subsets $S \subseteq A$ and $T \subseteq I$ with $|S| = N$ and $|T| = M$. Let D be the distribution on $S \cup T$ where each element of S has probability ρ and each element of T has probability $(1 - \rho N)/M = (1 - \rho N)/\lceil (1/\rho) - N \rceil = (1 - \rho N)/\lceil (1 - \rho N)/\rho \rceil \leq \rho$. Then $\Pr[w \in A \mid w \leftarrow D] \geq \rho N \geq 1 - \epsilon$, so D is a (ϵ, ρ) -improvising distribution. As above there is an expected finite-time probabilistic algorithm sampling from D , and this is an (ϵ, ρ) -improviser.

(c) \Rightarrow (a): Immediate. ◀

► **Remark.** In fact, whenever \mathcal{C} is feasible, the construction in the proof of Theorem 3.1 gives an improviser which works in nearly the most trivial possible way: it has two finite lists S and T , flips a (biased) coin to decide which list to use, and then returns an element of that list uniformly at random.

A consequence of this characterization is that when there are infinitely-many admissible improvisations, there is an improviser with zero error probability:

► **Corollary 3.2.** If A is infinite, $(\mathcal{I}, \alpha, 0, \rho)$ is feasible for any $\rho \in (0, 1] \cap \mathbb{Q}$.

In addition to giving conditions for feasibility, Theorem 3.1 yields an algorithm which is guaranteed to find an improviser for any feasible CI problem.

► **Corollary 3.3.** *If \mathcal{C} is feasible, an improviser for \mathcal{C} may be found by an effective procedure.*

Proof. The sets I and A are clearly computably enumerable, since α is computable. We enumerate I and A until enough elements are found to perform the construction in Theorem 3.1. Since \mathcal{C} is feasible, the theorem ensures this search will terminate. ◀

We cannot give an upper bound on the time needed by this algorithm without knowing something about the admissibility predicate α . Therefore although as noted in the remark above whenever there are improvisers at all there is one of a nearly-trivial form, actually finding such an improviser could be difficult. In fact, it could be faster to generate an improviser which is *not* of this form, as seen for example in Sec. 4.

► **Corollary 3.4.** *The set of feasible CI instances is computably enumerable but not computable.*

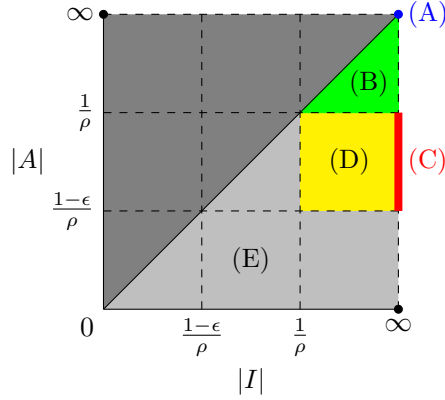
Proof. Enumerability follows immediately from the previous Corollary. If checking whether \mathcal{C} is feasible were decidable, then so would be checking if $|A| \geq (1 - \epsilon)/\rho$, but this is undecidable since α can be an arbitrary computable predicate. ◀

4 Finite-Memory Admissibility Predicates

In order to bound the time needed to find an improviser, we must constrain the admissibility predicate α . Perhaps the simplest type of admissibility predicate is one which can be computed by a DFA, i.e., one such that there is some DFA \mathcal{D} which accepts a word $w \in \Sigma^*$ iff $\alpha(w) = 1$. This captures the notion of a *finite-memory* admissibility predicate, where admissibility of a word can be determined by scanning the word left-to-right, only being able to remember a finite number of already-seen symbols. An example of a finite-memory predicate α is one such that $\alpha(w) = 1$ iff each subword of w of a fixed constant length satisfies some condition. By the pumping lemma, such predicates have the property that continually repeating some section of a word can produce an infinite family of improvisations, which could be a disadvantage if looking for “creative”, non-repetitive improvisations. However, in applications such as music we impose a maximum length on improvisations, so this is not an issue.

► **Example 4.1 (Factor Oracles).** Recall that one way of measuring the divergence of an improvisation w generated by the factor oracle F built from a word w_{ref} is by counting the number of non-direct transitions that w causes F to take. Since DFAs cannot count without bound, we can use a sliding window of some finite size k . Then our admissibility predicate α can be that at any point as F processes w , the number of the previous k transitions which were non-direct lies in some interval $[\ell, h]$ with $0 \leq \ell \leq h \leq k$. This predicate can be encoded as a DFA of size $O(|F| \cdot 2^k)$ (see the Appendix for details). The size of the automaton grows exponentially in the size of the window, but for small windows it can be reasonable.

When the admissibility predicate is finite-memory and the automaton \mathcal{I} is a DFA, there is an efficient procedure to test if an improviser exists and synthesize one if so. The construction is similar to that of Theorem 3.1, but avoids explicit enumeration of all improvisations to be put in the range of the improviser. To avoid enumeration we use a classic method of uniformly sampling from the language of a DFA \mathcal{D} (see for example [13, 10]). The next few lemmas summarize the results we need, proofs being given in the Appendix for completeness. The first step is to determine the size of the language.



■ **Figure 2** Cases for Theorem 4.5. The dark gray region cannot occur.

► **Lemma 4.2.** *If \mathcal{D} is a DFA, $|L(\mathcal{D})|$ can be computed in polynomial time.*

Once we know the size of $L(\mathcal{D})$ we can efficiently sample from it, handling infinite languages by sampling from a finite subset of a desired size.

► **Lemma 4.3.** *There is a polynomial $p(x, y)$ such that for any $N \in \mathbb{N}$ and DFA \mathcal{D} with infinite language, there is a probabilistic algorithm S which uniformly samples from a subset of $L(\mathcal{D})$ of size N in expected time at most $p(|\mathcal{D}|, \log N)$, and which can be constructed in the same time.*

► **Lemma 4.4.** *There is a polynomial $q(x)$ such that for any DFA \mathcal{D} with finite language, there is a probabilistic algorithm S which uniformly samples from $L(\mathcal{D})$ in expected time at most $q(|\mathcal{D}|)$, and which can be constructed in the same time.*

Using these sampling techniques, we have the following:

► **Theorem 4.5.** *The class of CI instances \mathcal{C} where \mathcal{I} is a DFA and α is computable by a DFA has a polynomial-time improvisation scheme.*

Proof. The proof considers five cases. We first define some notation. Let \mathcal{D} denote the DFA giving α . Letting \mathcal{A} be the product of \mathcal{I} and \mathcal{D} , we have $A = L(\mathcal{A})$. This product can be computed in polynomial time since the automata are both DFAs, and $|A|$ is polynomial in $|\mathcal{C}|$ and $|\mathcal{D}|$. In some of the cases below we will also use a DFA \mathcal{B} which is the synchronous product of \mathcal{I} and the complement of \mathcal{A} . Clearly $L(\mathcal{B}) = I \setminus A$, and the size of \mathcal{B} and the time needed to construct it are also polynomial in $|\mathcal{C}|$ and $|\mathcal{D}|$.

Next we compute $|A| = |L(\mathcal{A})|$ and $|I| = |L(\mathcal{I})|$ in polynomial time using Lemma 4.2. There are now several cases (illustrated in Figure 2):

- (A) $|A| = \infty$: Applying Lemma 4.3 to \mathcal{A} with $N = \lceil 1/\rho \rceil$, we obtain a probabilistic algorithm S which uniformly samples from a subset of $L(\mathcal{A}) = A$ of size $\lceil 1/\rho \rceil$. Since $1/\lceil 1/\rho \rceil \leq \rho$, we have that S is a $(0, \rho)$ -improviser and return it.
- (B) $1/\rho \leq |A| < \infty$: Applying Lemma 4.4 to \mathcal{A} , we obtain a probabilistic algorithm S which uniformly samples from $L(\mathcal{A}) = A$. Since $1/|A| \leq \rho$, we have that S is a $(0, \rho)$ -improviser and return it.
- (C) $(1 - \epsilon)/\rho \leq |A| < 1/\rho$ and $|I| = \infty$: Applying Lemma 4.4 to \mathcal{A} we obtain S as in the previous case. Defining $M = \lceil 1/\rho \rceil - |A|$, we have $\infty = |L(\mathcal{B})| > M \geq 1$. Applying Lemma 4.3 to \mathcal{B} with $N = M$ yields a probabilistic algorithm S' which

uniformly samples from a subset of $L(\mathcal{B}) = I \setminus A$ of size M . Let G be a probabilistic algorithm which with probability $\rho|A|$ executes S , and otherwise executes S' . Then since $L(\mathcal{A}) = A$ and $L(\mathcal{B}) = I \setminus A$ are disjoint, every word generated by G has probability either $(\rho|A|)/|A| = \rho$ (if it is in A) or $(1 - \rho|A|)/M = (1 - \rho|A|)/\lceil(1 - \rho|A|)/\rho\rceil \leq \rho$ (if it is in $I \setminus A$). Also, G outputs a member of A with probability $\rho|A| \geq 1 - \epsilon$, so G is an (ϵ, ρ) -improviser and we return it.

- (D) $(1 - \epsilon)/\rho \leq |A| < 1/\rho \leq |I| < \infty$: As in the previous case, except obtaining S' by applying Lemma 4.4 to \mathcal{B} . Since $|I| \geq \lceil 1/\rho \rceil$, we have $|L(\mathcal{B})| = |I \setminus A| \geq M$ and so G as constructed above is an (ϵ, ρ) -improviser.
- (E) $|I| < 1/\rho$ or $|A| < (1 - \epsilon)/\rho$: By Theorem 3.1, \mathcal{C} is not feasible, so we return \perp .

This procedure takes time polynomial in $|\mathcal{I}|$, $|\mathcal{D}|$, and $\log(1/\rho)$, so it is polynomial-time. Also, a fixed polynomial in these quantities bounds the expected runtime of the generated improviser, so the procedure is a polynomial-time improvisation scheme. \blacktriangleleft

Running Example. Recall that for our running example $\mathcal{C} = (\mathcal{I}, \alpha, 1/4, 1/4)$, we have $I = \{000, 001, 010, 100, 101\}$ and $A = \{000, 001, 101\}$. Since $|A| = 3$ and $|I| = 5$, we are in case (D) of Theorem 4.5. So our scheme uses Lemma 4.4 to obtain S and S' uniformly sampling from A and $I \setminus A = \{010, 100\}$ respectively. It returns a probabilistic algorithm G that executes S with probability $\rho|A| = 3/4$ and otherwise executes S' . So G returns 000, 001, and 101 with probability $1/4$ each, and 010 and 100 with probability $1/8$ each. The output distribution of G satisfies our conditions, so it is an improviser for \mathcal{C} .

5 More Complex Automata

While counting the language of a DFA is easy, in the case of an NFA it is much more difficult, and so there are unlikely to be polynomial-time improvisation schemes for more complex automata. Let \mathcal{N}_1 and \mathcal{N}_2 be the classes of CI instances where \mathcal{I} or α respectively are given by an NFA, and the other is given by a DFA. Then denoting by \mathcal{N} either of these classes, we have (deferring full proofs from this section to the Appendix):

► **Theorem 5.1.** *Determining whether $\mathcal{C} \in \mathcal{N}$ is feasible is #P-hard.*

Proof sketch. The problem of counting the language of an NFA, which is #P-hard [14], is polynomially reducible to that of checking if $\mathcal{C} \in \mathcal{N}$ is feasible. \blacktriangleleft

► **Remark.** Determining feasibility of \mathcal{N} -instances is not a counting problem, so it is not #P-complete, but it is clearly in $\mathsf{P}^{\#P}$: we construct the automata \mathcal{I} and \mathcal{A} as in Theorem 4.5 (now they can be NFAs), count their languages using #P, and apply Theorem 3.1.

► **Corollary 5.2.** *If there is a polynomial-time improvisation scheme for \mathcal{N} , then $\mathsf{P} = \mathsf{P}^{\#P}$.*

This result indicates that in general, the control improvisation problem is probably intractable in the presence of NFAs. Some special cases could still be handled in practice: for example, if the NFA is very small it could be converted to a DFA. Another tractable case is where although one of \mathcal{I} or \mathcal{A} (as in Theorem 4.5) is an NFA, it has infinite language (this can clearly be detected in polynomial time). If \mathcal{A} is an NFA with infinite language we can use case (A) of the proof of Theorem 4.5, since an NFA can be pumped in the same way as a DFA. If instead \mathcal{A} is a DFA with finite language but \mathcal{I} is an NFA with infinite language, one of the other cases (B), (C), or (E) applies, and in case (C) we can sample $I \setminus A$ by pumping

\mathcal{I} enough to ensure we get a string longer than any accepted by \mathcal{A} . Table 1 in Section 7 summarizes these cases.

For still more complex automata, the CI problem becomes even harder. In fact, it is impossible if we allow either \mathcal{I} or α to be given by a PFA. Let \mathcal{P}_1 and \mathcal{P}_2 be the classes of CI instances where each of these respectively are given by a PFA, and the other is given by a DFA. Then letting \mathcal{P} be either of these classes, we have:

► **Theorem 5.3.** *Determining whether $\mathcal{C} \in \mathcal{P}$ is feasible is undecidable.*

Proof sketch. Checking the feasibility of \mathcal{C} amounts to counting the language of a PFA, but determining whether the language of a PFA is empty is undecidable [17, 9]. ◀

6 Symbolic Techniques

Previously we have assumed that the automata defining a control improvisation problem were given explicitly. However, in practice there may be insufficient memory to store full transition tables, in which case an implicit representation is required. This prevents us from using the polynomial-time improvisation scheme of Theorem 4.5, so we must look for alternate methods. These will depend on the type of implicit representation used. We focus on representations of DFAs and NFAs by propositional formulae, as used for example in bounded model checking [4].

► **Definition 6.1.** A *symbolic automaton* is a transition system over states $S \subseteq \{0, 1\}^n$ and inputs $\Sigma \subseteq \{0, 1\}^m$ represented by:

- a formula $\text{init}(\bar{x})$ which is true iff $\bar{x} \in \{0, 1\}^n$ is an initial state,
- a formula $\text{acc}(\bar{x})$ which is true iff $\bar{x} \in \{0, 1\}^n$ is an accepting state, and
- a formula $\delta(\bar{x}, \bar{a}, \bar{y})$ which is true iff there is a transition from $\bar{x} \in \{0, 1\}^n$ to $\bar{y} \in \{0, 1\}^n$ on input $\bar{a} \in \{0, 1\}^m$.

A symbolic automaton accepts words in Σ^* according to the usual definition for NFAs.

Given a symbolic automaton, it is straightforward to generate a formula whose models correspond, for example, to accepting paths of at most a given length (see [4] for details). A SAT solver can then be used to find such a path. We refer to the length of the longest simple accepting path as the *diameter* of the automaton. This will be an important parameter in the runtime of our algorithms. In some cases an upper bound on the diameter is known ahead of time: for example, if we only want improvisations of up to some maximum length, and have encoded that constraint in \mathcal{I} . If the diameter is not known, it can be found iteratively with SAT queries asserting the existence of a simple accepting path of length n , increasing n until we find no such path exists. The diameter could be exponentially large compared to the symbolic representation, but this is a worst-case scenario.

Our approach for solving the control improvisation problem with symbolic automata will be to adapt the procedure of Theorem 4.5, replacing the counting and sampling techniques used there with ones that work on symbolic automata. For language size estimation we use the following:

► **Lemma 6.2.** *If \mathcal{S} is a symbolic automaton with diameter D , for any $\tau, \delta > 0$ we can compute an estimate of $|L(\mathcal{S})|$ accurate to within a factor of $1 + \tau$ in time polynomial in $|\mathcal{S}|$, D , $1/\tau$, and $\log(1/\delta)$ relative to an NP oracle.*

Sampling from infinite languages can be done by a direct adaptation of the method for explicit DFAs in Lemma 4.3.

► **Lemma 6.3.** *There is a polynomial $p(x, y, z)$ such that for any $N \in \mathbb{N}$ and symbolic automaton \mathcal{Y} with infinite language and diameter D , there is a probabilistic oracle algorithm S^{NP} which uniformly samples from a subset of $L(\mathcal{Y})$ of size N in expected time at most $p(|\mathcal{Y}|, D, \log N)$ and which can be constructed in the same time.*

To sample from a finite language, we use techniques for almost-uniform generation of models of propositional formulae. In theory uniform sampling can be done exactly using a SAT solver [3], but the only algorithms which work in practice are *approximate* uniform generators such as UniGen [7]. This algorithm guarantees that the probability of returning any given model is within a factor of $1 + \tau$ of the uniform probability, for any given $\tau > 6.84$ (the constant is for technical reasons specific to UniGen). UniGen can also do projection sampling, i.e., sampling where two models are considered identical if they agree on the set of variables being projected onto. Henceforth, for simplicity, we will assume we have a generic almost-uniform generator that can do projection, and will ignore the $\tau > 6.84$ restriction imposed by UniGen (although we might want to abide by this in practice in order to be able to use the fastest available algorithm). We assume that the generator runs in time polynomial in $1/\tau$ and the size of the given formula relative to an NP oracle, and succeeds with at least some fixed constant probability.

► **Lemma 6.4.** *There is a polynomial $q(x, y, z)$ such that for any $\tau > 0$ and symbolic automaton \mathcal{Y} with finite language and diameter D , there is a probabilistic oracle algorithm S^{NP} which samples from $L(\mathcal{S})$ uniformly up to a factor of $1 + \tau$ in expected time at most $q(|\mathcal{Y}|, D, 1/\tau)$, and which can be constructed in the same time.*

Now we can put these methods together to get a version of Theorem 4.5 for symbolic automata. The major differences are that this scheme requires an NP oracle, has some probability of failure (which can be specified), and returns an improviser with a slightly sub-optimal value of ρ . The proof generally follows that of Theorem 4.5, so we only sketch the differences here (see the Appendix for a full proof).

► **Theorem 6.5.** *There is a procedure that given any CI problem \mathcal{C} where \mathcal{I} and α are given by symbolic automata with diameter at most D , and any $\epsilon \in [0, 1]$, $\rho \in (0, 1]$, and $\tau, \delta > 0$, if \mathcal{C} is $(\epsilon, \rho/(1 + \tau))$ -feasible returns an $(\epsilon, (1 + \tau)^2(1 + \epsilon)\rho)$ -improviser with probability at least $1 - \delta$. Furthermore, the procedure and the improvisers it generates run in expected time given by some fixed polynomial in $|\mathcal{C}|$, D , $1/\tau$, and $\log(1/\delta)$ relative to an NP oracle.*

Proof sketch. We first compute estimates E_A and E_I of $|A|$ and $|I|$ respectively using Lemma 6.2. Then we break into cases as in Theorem 4.5:

- (A) $E_A = \infty$: As in case (A) of Theorem 4.5, using Lemma 6.3 in place of Lemma 4.3. We obtain a $(0, \rho)$ -improviser.
- (B) $1/\rho \leq E_A < \infty$: As in case (B) of Theorem 4.5, using Lemma 6.4 in place of Lemma 4.4. Since we can do only approximate counting and sampling, we obtain a $(0, (1 + \tau)^2 \rho)$ -improviser.
- (C) $(1 - \epsilon)/\rho \leq E_A < 1/\rho$ and $E_I = \infty$: As in case (C) of Theorem 4.5, using Lemmas 6.3 and 6.4 in place of Lemmas 4.3 and 4.4. Our use of approximate counting/sampling means we obtain only an $(\epsilon, (1 + \tau)^2 \rho)$ -improviser.
- (D) $(1 - \epsilon)/\rho \leq E_A < 1/\rho \leq E_I < \infty$: We cannot use the procedure in case (D) of Theorem 4.5, since it may generate an element of $L(\mathcal{B})$ with too high probability if our estimate E_A is sufficiently small. Instead we sample almost-uniformly from $L(\mathcal{A})$ with probability ϵ , and from $L(\mathcal{I})$ with probability $1 - \epsilon$. This yields an $(\epsilon, (1 + \tau)^2(1 + \epsilon)\rho)$ -improviser.

\mathcal{I}	α	DFA	NFA		PFA
			$L(\mathcal{A}) = \infty$	$L(\mathcal{A}) < \infty$	
DFA		poly-time		#P-hard	
NFA	$L(\mathcal{I}) = \infty$				
	$L(\mathcal{I}) < \infty$	#P-hard	-		
PFA		undecidable			

■ **Table 1** Complexity of the control improvisation problem when \mathcal{I} and α are given by various different types of automata. The cell marked ‘-’ is impossible since $L(\mathcal{A}) \subseteq L(\mathcal{I})$.

(E) $E_I < 1/\rho$ or $E_A < (1 - \epsilon)/\rho$: We return \perp .

If \mathcal{C} is $(\epsilon, \rho/(1 + \tau))$ -feasible, case (E) happens with probability less than δ by Theorem 3.1. Otherwise, we obtain an $(\epsilon, (1 + \tau)^2(1 + \epsilon)\rho)$ -improviser. ◀

Therefore, it is possible to *approximately* solve the control improvisation problem when the automata are given by a succinct propositional formula representation. This allows working with general NFAs, and very large automata that cannot be stored explicitly, but comes at the cost of using a SAT solver (perhaps not a heavy cost given the dramatic advances in the capacity of SAT solvers) and possibly having to increase ρ by a small factor.

7 Conclusion

In this paper, we introduced control improvisation, the problem of creating improvisers that randomly generate variants of words in the languages of automata. We gave precise conditions for when improvisers exist, and investigated the complexity of finding improvisers for several major classes of automata. In particular, we showed that the control improvisation problem for DFAs can be solved in polynomial time, while it is intractable in most cases for NFAs and undecidable for PFAs. These results are summarized in Table 1. Finally, we studied the case where the automata are presented symbolically instead of explicitly, and showed that the control improvisation problem can still be solved approximately using SAT solvers.

One interesting direction for future work would be to find other tractable cases of the control improvisation problem deriving from finer structural properties of the automata than just determinism. Extensions of the theory to other classes of formal languages, for instance context-free languages represented by pushdown automata or context-free grammars, are also worthy of study. Finally, we are investigating further applications, particularly in the areas of testing, security, and privacy.

Acknowledgements The first three authors dedicate this paper to the memory of the fourth author, David Wessel, who passed away while it was being written. We would also like to thank Ben Caulfield, Orna Kupferman, Markus Rabe, and the anonymous reviewers for their helpful comments. This work is supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1106400, by the NSF Expeditions grant CCF-1139138, and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

- 1 Gérard Assayag, Georges Bloch, Marc Chemillier, Benjamin Lévy, and Shlomo Dubnov. OMax. <http://repmus.ircam.fr/omax/home>, 2012.
- 2 Gérard Assayag and Shlomo Dubnov. Using factor oracles for machine improvisation. *Soft Comput.*, 8(9):604–610, 2004.
- 3 Mihir Bellare, Oded Goldreich, and Erez Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 2000.
- 4 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.
- 5 Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- 6 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming*, pages 200–216. Springer, 2013.
- 7 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *51st Design Automation Conference*, pages 1–6. ACM, 2014.
- 8 Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. Constructing factor oracles. In *In Proceedings of the 3rd Prague Stringology Conference*, 2003.
- 9 Anne Condon and Richard J. Lipton. On the complexity of space bounded interactive proofs. In *30th Annual Symposium on Foundations of Computer Science*, pages 462–467. IEEE, 1989.
- 10 Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, and Sylvain Peyronnet. Uniform random sampling of traces in very large models. In *1st International Workshop on Random Testing*, pages 10–19. ACM, 2006.
- 11 Alexandre Donzé, Rafael Valle, Ilge Akkaya, Sophie Libkind, Sanjit A. Seshia, and David Wessel. Machine improvisation with formal specifications. In *Proceedings of the 40th International Computer Music Conference (ICMC)*, September 2014.
- 12 Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE Future of Software Engineering)*. IEEE, May 2014.
- 13 Timothy Hickey and Jacques Cohen. Uniform random generation of strings in a context-free language. *SIAM Journal on Computing*, 12(4):645–655, 1983.
- 14 Sampath Kannan, Z. Sweedyk, and Steve Mahaney. Counting and random generation of strings in regular languages. In *Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 551–557. SIAM, 1995.
- 15 Stéphane Lafortune. Personal Communication, 2015.
- 16 Edward A. Lee. Personal Communication, 2013.
- 17 Masakazu Nasu and Namio Honda. Mappings induced by PGSM-mappings and some recursively unsolvable problems of finite probabilistic automata. *Information and Control*, 15(3):250–273, 1969.
- 18 Michael O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.
- 19 R. Rowe. *Machine Musicianship*. MIT Press, 2001.
- 20 Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

A Proofs

► **Example 4.1** (Factor Oracles). The factor oracle-based admissibility predicate α described above can be encoded as a DFA of size $O(|F| \cdot 2^k)$ as follows: we have a copy of F , denoted F_s , for every string $s \in \{0, 1\}^k$, each bit of s indicating whether the corresponding previous transition (out of the last k) was non-direct. As each new symbol is processed, we execute the current copy of F as usual, but move to the appropriate state of the copy of F corresponding to the new k -transition history, i.e., if we were in F_s , we move to F_t where t consists of the last $k - 1$ bits of s followed by a 0 if the transition we took was direct and a 1 otherwise. Making the states of F_s accepting iff the number of 1s in s is in $[\ell, h]$, this automaton represents α as desired.

► **Lemma 4.2.** *If \mathcal{D} is a DFA, $|L(\mathcal{D})|$ can be computed in polynomial time.*

Proof. First we prune irrelevant states unreachable from the initial state or from which no accepting state can be reached (this pruning can clearly be done in polynomial time). If the resulting graph contains a cycle (also detectable in polynomial time), we return ∞ . Otherwise \mathcal{D} is a DAG with multiple edges, and every sink is an accepting state. For each accepting state s we add a new vertex and an edge to it from s . Then there is a one-to-one correspondence between accepting words of \mathcal{D} and paths from the initial state to a sink. Now we can compute for each vertex v the number of paths p_v from it to a sink using the usual linear-time DAG algorithm (traversal in reverse topological order) modified slightly to handle multiple edges. We return p_v with v the initial state. ◀

► **Lemma 4.3.** *There is a polynomial $p(x, y)$ such that for any $N \in \mathbb{N}$ and DFA \mathcal{D} with infinite language, there is a probabilistic algorithm S which uniformly samples from a subset of $L(\mathcal{D})$ of size N in expected time at most $p(|\mathcal{D}|, \log N)$, and which can be constructed in the same time.*

Proof. Having pruned \mathcal{D} as in Lemma 4.2, since $L(\mathcal{D})$ is infinite there must be some state s of \mathcal{D} such that

- there is a word $x \in \Sigma^*$ which takes \mathcal{D} from its initial state to s ,
- there is a nonempty word $y \in \Sigma^*$ which takes \mathcal{D} from s to itself, and
- there is a word $z \in \Sigma^*$ which takes \mathcal{D} from s to an accepting state.

We can find $x, y, z \in \Sigma^*$ as above with $|x|, |y|, |z| \leq |\mathcal{D}|$, in time polynomial in $|\mathcal{D}|$. Then we have $xy^n z \in L(\mathcal{D})$ for any $n \in \mathbb{N}$. We form a probabilistic algorithm S which acts as follows: it prints x , then picks an integer uniformly at random from $[0, N - 1]$ and prints that many copies of y , before finally printing z . Clearly the output of S is a uniform sample from a subset of $L(\mathcal{D})$ of size N . Constructing S takes time polynomial in $|\mathcal{D}|$ (as this bounds the sizes of x , y , and z) and $\log N$, and S runs in expected time bounded by a fixed polynomial in these values. ◀

► **Lemma 4.4.** *There is a polynomial $q(x)$ such that for any DFA \mathcal{D} with finite language, there is a probabilistic algorithm S which uniformly samples from $L(\mathcal{D})$ in expected time at most $q(|\mathcal{D}|)$, and which can be constructed in the same time.*

Proof. Prune \mathcal{D} and compute the path counts p_v as in Lemma 4.2. To every edge (u, v) in \mathcal{D} assign the weight p_v/p_u . It is clear that at every vertex the sum of the weights of the outgoing edges is 1 (unless the vertex is a sink). We prove by induction along reverse topological order that treating these weights as transition probabilities, starting from any state u and talking a random walk until a sink is reached we obtain a uniform distribution

over all paths from u to a sink. If u is a sink this holds trivially. If u has a nonempty set of children S , then by the inductive hypothesis for every $v \in S$ starting a walk at v gives a uniform distribution over the p_v paths from v to a sink. Therefore the probability of following any such path starting at u is $(p_v/p_u) \cdot (1/p_v) = 1/p_u$. So the result holds by induction. In particular, if we start from the initial state we obtain a uniform distribution over all paths to a sink, and thus a uniform distribution over $L(\mathcal{D})$. Since all probabilities are rational with denominators bounded by $|\Sigma|^{|\mathcal{D}|}$, this walk can be performed by a probabilistic algorithm S of size polynomial in $|\mathcal{D}|$, with expected time bounded by a fixed polynomial in $|\mathcal{D}|$. Then S returns a uniform sample from $L(\mathcal{D})$, and it can be constructed in time polynomial in $|\mathcal{D}|$. ◀

► **Theorem 5.1.** *Determining whether $\mathcal{C} \in \mathcal{N}$ is feasible is #P-hard.*

Proof. We prove this for \mathcal{N}_1 — the other case is analogous. As shown in [14], the problem of determining $|L(\mathcal{M}) \cap \Sigma^m|$ given an NFA \mathcal{M} over an alphabet Σ and $m \in \mathbb{N}$ in unary is #P-complete. We give a polynomial-time (Cook) reduction from this problem to checking feasibility of a CI instance in \mathcal{N}_1 .

As noted in [14], we can in polynomial time (in m , which is acceptable since m is given in unary) construct an NFA \mathcal{M}' such that $|L(\mathcal{M}')| = |L(\mathcal{M}) \cap \Sigma^m|$. Then we construct the trivial DFA \mathcal{T} accepting all of Σ^* , and consider the CI instances $\mathcal{C}_N = (\mathcal{M}', \mathcal{T}, 0, 1/N)$. Clearly for these instances we have $I = A = L(\mathcal{M}')$. By Theorem 3.1, \mathcal{C}_N is feasible iff $|L(\mathcal{M}) \cap \Sigma^m| = |L(\mathcal{M}')| \geq N$, and since $\mathcal{C}_N \in \mathcal{N}_1$ for any $N \in \mathbb{N}$ we can determine whether this is the case. Since $|L(\mathcal{M}) \cap \Sigma^m| \leq |\Sigma|^m$, using binary search we can find the exact value of $|L(\mathcal{M}) \cap \Sigma^m|$ with polynomially-many such queries. ◀

► **Theorem 5.3.** *Determining whether $\mathcal{C} \in \mathcal{P}$ is feasible is undecidable.*

Proof. We prove this for \mathcal{P}_1 , the other case being similar. Given a PFA \mathcal{A} with cut-point p over an alphabet Σ with at least two symbols, determining whether the language of \mathcal{A} is empty (i.e. whether it accepts no words with probability greater than p) is undecidable [17, 9]. For any $N > 0$, we can construct a PFA \mathcal{A}' by adding new states and deterministic transitions to \mathcal{A} so that there are exactly N words taking \mathcal{A}' from its initial state to the initial state of \mathcal{A} , and any word which does not have one of these as a prefix causes \mathcal{A}' to reject with probability 1. Then $L(\mathcal{A}')$ is $L(\mathcal{A})$ with one of the N prefixes added to each word. Therefore $|L(\mathcal{A}')| \geq N$ iff $L(\mathcal{A}) \neq \emptyset$, and so it is undecidable to determine whether the language of a PFA has at least N elements.

Constructing the trivial DFA \mathcal{T} accepting all of Σ^* , the CI instance $\mathcal{C} = (\mathcal{A}, \mathcal{T}, 0, 1/N)$ satisfies $I = A = L(\mathcal{A})$. By Theorem 3.1, \mathcal{C} is feasible iff $|L(\mathcal{A})| \geq N$. Since checking this latter condition is undecidable, so is determining feasibility of \mathcal{P}_1 -instances. ◀

► **Lemma 6.2.** *If \mathcal{S} is a symbolic automaton with diameter D , for any $\tau, \delta > 0$ we can compute an estimate of $|L(\mathcal{S})|$ accurate to within a factor of $1 + \tau$ in time polynomial in $|\mathcal{S}|$, D , $1/\tau$, and $\log(1/\delta)$ relative to an NP oracle.*

Proof. Recall that the algorithm in Lemma 4.2 detected accepting cycles by finding words x, y, z taking the automaton to some state s , from s to s along at least one transition, and to an accepting state respectively. If we impose the additional constraint $|x|, |y|, |z| \leq n$, then we can check the existence of such words with a single SAT query. So to test whether $L(\mathcal{S})$ is infinite, we use one query for each $n \leq D$: since D is the diameter, we are guaranteed to find an accepting cycle if one exists. Thus if any query is satisfiable, we return ∞ .

If instead all the queries fail, then all words in $L(\mathcal{S})$ have length at most D . So the SAT query ϕ for accepting paths of length at most D in fact matches every accepting path. Models of this formula then correspond to accepting words if we project onto the input variables (i.e. ignore the values of the variables which encode states). Therefore $|L(\mathcal{S})|$ is equal to the number of models of ϕ after projection. The general problem of counting models of a propositional formula (even without projection) is $\#P$ -complete, but using the SAT solver we can get probabilistic bounds. An approximate model counter such as **ApproxMC** [6] can return an estimate of the number of models of ϕ which is accurate to within a factor of $1 + \tau$ with probability at least $1 - \delta$. In fact **ApproxMC** can be easily modified to do projection counting (see [7]), giving us the required estimate of $|L(\mathcal{S})|$.

The first stage of this process clearly takes time polynomial in $|\mathcal{S}|$ and D relative to the oracle. **ApproxMC** runs in time polynomial in $|\phi| = O(D|\mathcal{S}|)$, $1/\tau$, and $\log(1/\delta)$ relative to the oracle, so this procedure does as well. \blacktriangleleft

► **Lemma 6.3.** *There is a polynomial $p(x, y, z)$ such that for any $N \in \mathbb{N}$ and symbolic automaton \mathcal{Y} with infinite language and diameter D , there is a probabilistic oracle algorithm S^{NP} which uniformly samples from a subset of $L(\mathcal{Y})$ of size N in expected time at most $p(|\mathcal{Y}|, D, \log N)$ and which can be constructed in the same time.*

Proof. We look for an accepting cycle using the method in Lemma 6.2. One will be found since $|L(\mathcal{Y})|$ is infinite, and then we can pump it to get N different words just as in Lemma 4.3. \blacktriangleleft

► **Lemma 6.4.** *There is a polynomial $q(x, y, z)$ such that for any $\tau > 0$ and symbolic automaton \mathcal{Y} with finite language and diameter D , there is a probabilistic oracle algorithm S^{NP} which samples from $L(\mathcal{S})$ uniformly up to a factor of $1 + \tau$ in expected time at most $q(|\mathcal{Y}|, D, 1/\tau)$, and which can be constructed in the same time.*

Proof. As noted in Lemma 6.2, since the language is finite every word in it has length at most D . Constructing the formula ϕ from that lemma, once we project onto the input variables there is a one-to-one correspondence between accepting words and models of ϕ . So we need to almost-uniformly generate projected models of a propositional formula. We use an almost-uniform generator as described above, whose runtime will be polynomial in $|\phi| = O(D|\mathcal{Y}|)$ and $1/\tau$ relative to the oracle. \blacktriangleleft

► **Theorem 6.5.** *There is a procedure that given any CI problem \mathcal{C} where \mathcal{I} and α are given by symbolic automata with diameter at most D , and any $\epsilon \in [0, 1]$, $\rho \in (0, 1]$, and $\tau, \delta > 0$, if \mathcal{C} is $(\epsilon, \rho/(1 + \tau))$ -feasible returns an $(\epsilon, (1 + \tau)^2(1 + \epsilon)\rho)$ -improviser with probability at least $1 - \delta$. Furthermore, the procedure and the improvisers it generates run in expected time given by some fixed polynomial in $|\mathcal{C}|$, D , $1/\tau$, and $\log(1/\delta)$ relative to an NP oracle.*

Proof. The procedure begins by deriving the symbolic representations of \mathcal{A} and \mathcal{B} (the product of \mathcal{I} and the complement of \mathcal{A}). Next we estimate $|A| = |L(\mathcal{A})|$ and $|I| = |L(\mathcal{I})|$ using Lemma 6.2 with a confidence of $(1 - \delta)^{1/2}$. Then with probability at least $1 - \delta$, both these estimates are within a factor of $1 + \tau$ of the true values. We assume this is the case for the rest of the proof, making no guarantees otherwise. We now break into the same cases as Theorem 4.5, using our estimates E_A and E_I of $|A|$ and $|I|$ respectively.

(A) $E_A = \infty$: In this case we must have $|A| = \infty$. We proceed as in case (A) of Theorem 4.5, but using Lemma 6.3 with $N = \lceil 1/\rho \rceil$ in place of Lemma 4.3 to obtain a $(0, \rho)$ -improviser.

- (B) $1/\rho \leq E_A < \infty$: Then $1/\rho(1+\tau) \leq E_A/(1+\tau) \leq |A|$. We proceed as in case (B) of Theorem 4.5, using Lemma 6.4 in place of Lemma 4.4. Since we are using an almost-uniform generator instead of a uniform one, some words could have probability as high as $(1+\tau)/|A| \leq (1+\tau)^2\rho$, and so this gives us a $(0, (1+\tau)^2\rho)$ -improviser.
- (C) $(1-\epsilon)/\rho \leq E_A < 1/\rho$ and $E_I = \infty$: Then $(1-\epsilon)/\rho(1+\tau) \leq E_A/(1+\tau) \leq |A|$, and $|I| = \infty$. We proceed along the same lines as case (C) of Theorem 4.5. We use Lemma 6.4 as in the previous case to generate a probabilistic algorithm S almost-uniformly sampling from $L(\mathcal{A})$. Defining $M = \lceil (1/\rho - E_A)/(1+\tau) \rceil$, we have $\infty = |L(\mathcal{B})| > M \geq 1$. We can use Lemma 6.3 in place of Lemma 4.3 to get a probabilistic algorithm S' uniformly sampling from a subset of $L(\mathcal{B})$ of size M . Let G be a probabilistic algorithm which with probability ρE_A executes S , and otherwise executes S' . Then since $L(\mathcal{A}) = A$ and $L(\mathcal{B}) = I \setminus A$ are disjoint, every word generated by G has probability either at most $(\rho E_A) \cdot (1+\tau)/|A| \leq (1+\tau)^2\rho$ (if it is in A) or at most $(1-\rho E_A)/M = (1-\rho E_A)/\lceil (1/\rho - E_A)/(1+\tau) \rceil = (1-\rho E_A)/[(1-\rho E_A)/(1+\tau)\rho] \leq (1+\tau)\rho$ (if it is in $I \setminus A$). Also G outputs a member of A with probability $\rho E_A \geq 1-\epsilon$, so G is an $(\epsilon, (1+\tau)^2\rho)$ -improviser and we return it.
- (D) $(1-\epsilon)/\rho \leq E_A < 1/\rho \leq E_I < \infty$: Then $(1-\epsilon)/\rho(1+\tau) \leq E_A/(1+\tau) \leq |A|$, and $1/\rho(1+\tau) \leq E_I/(1+\tau) \leq |I|$. As in the previous case, use Lemma 6.4 to produce a probabilistic algorithm S almost-uniformly sampling from $L(\mathcal{A})$. Since $L(\mathcal{I})$ is finite, we can use the same technique to get a probabilistic algorithm S' almost-uniformly sampling from $L(\mathcal{I})$. Let G be a probabilistic algorithm which with probability $1-\epsilon$ executes S , and otherwise executes S' . Then G generates each $w \in A$ with probability at most $[(1-\epsilon) \cdot (1+\tau)/|A|] + [\epsilon \cdot (1+\tau)/|I|] \leq (1+\tau)^2\rho + (1+\tau)^2\epsilon\rho = (1+\tau)^2(1+\epsilon)\rho$, and each $w \in I \setminus A$ with probability at most $\epsilon \cdot (1+\tau)/|I| \leq (1+\tau)^2\epsilon\rho$. Furthermore G outputs a member of A with probability at least $1-\epsilon$, so G is an $(\epsilon, (1+\tau)^2(1+\epsilon)\rho)$ -improviser and we return it.
- (E) $E_I < 1/\rho$ or $E_A < (1-\epsilon)/\rho$: It is possible that the problem is not $(\epsilon, \rho/(1+\tau))$ -feasible, so the procedure returns \perp .

In the cases where an almost-uniform generator is used, there is some constant probability that the generator will fail. If that happens, the improviser just runs the generator again: since the failure probability is a fixed constant, so is the expected number of repetitions needed, and thus the expected runtime of the improviser is just multiplied by an overall constant.

Now if \mathcal{C} is $(\epsilon, \rho/(1+\tau))$ -feasible, by Theorem 3.1 we have $1/\rho \leq |I|/(1+\tau) \leq E_I$ and $(1-\epsilon)/\rho \leq |A|/(1+\tau) \leq E_A$ with probability at least $1-\delta$. So with probability $1-\delta$ case (E) does not happen, and the procedure returns an $(\epsilon, (1+\tau)\rho)$ -improviser. \blacktriangleleft